Expi2Java
An Extensible Code Generator for Security Protocols
User Manual

Alex Busenius

May 11, 2011

1 Introduction

1.1 Features

Expi2java is a code generator for cryptographic protocols. It takes protocol specifications written in an extensible variant of the Spi calculus [AB05], configurations that provide the low-level information needed for code generation and produces interoperable Java code. The main features of Expi2java are:

- Compatibility with ProVerif [Bla08]
  - The same input language as in ProVerif with few extensions
  - Automatic conversion to the ProVerif format
- An expressive type system
  - Supports nested types through parametric polymorphism
  - Supports additional low-level information in the form of configurations
  - Prevents incorrect use of cryptographic primitives and configurations
- High customizability and extensibility
  - User-defined types, cryptographic primitives and low-level configurations
  - User-provided implementation classes
  - Code generation using code templates
- Interoperability of the generated code
  - Generated implementation uses Java cryptographic providers
  - Most common cryptographic primitives and data types are supported out of the box
  - Full control over the low-level data format

This manual describes the complete process of generating runnable code with Expi2java from the user's point of view. It does not describe all internal details of the type checker and code generator, since they are not needed for the normal user. Although the extension of the runtime library does require implementing new Java classes, the portion of the code that needs to be changed in this case is independent from the rest of the tool.

1.2 Outline

Section 2 gives a short overview about the design of Expi2java and the usual code generation workflow. The input language used to write the protocol specifications, as well as the low-level configurations and the types are described in Section 3. The command line options used by Expi2java are described in Section 4. Section 5 explains the code generation process on a simple "Hello World" example. Please note that there is also a more detailed tutorial explaining the code generation process on the example of the Perrig-Song protocol. Finally, Section 6 describes various ways of customizing and extending Expi2java.

2 Workflow

2.1 Design Overview

The overall design of the Expi2java framework is shown in Figure 1. There are two kinds of input files. The Expi file contains the protocol and the Exdef files contain the low-level information for the code generation. The input files are parsed and transformed into an internal in-memory representation. This data is then used as the input to the *type checker* that performs a consistency check. Afterwards, the data is passed to the code generator that produces Java code using code templates. The generated code uses a *runtime library*. The runtime library is a set of classes used to implement the behavior of the types, constructors and destructors in the generated code (see also Section 6.5). Additionally, a *pretty-printer* can be used to transfer the input files into the syntax that can be analyzed by ProVerif.

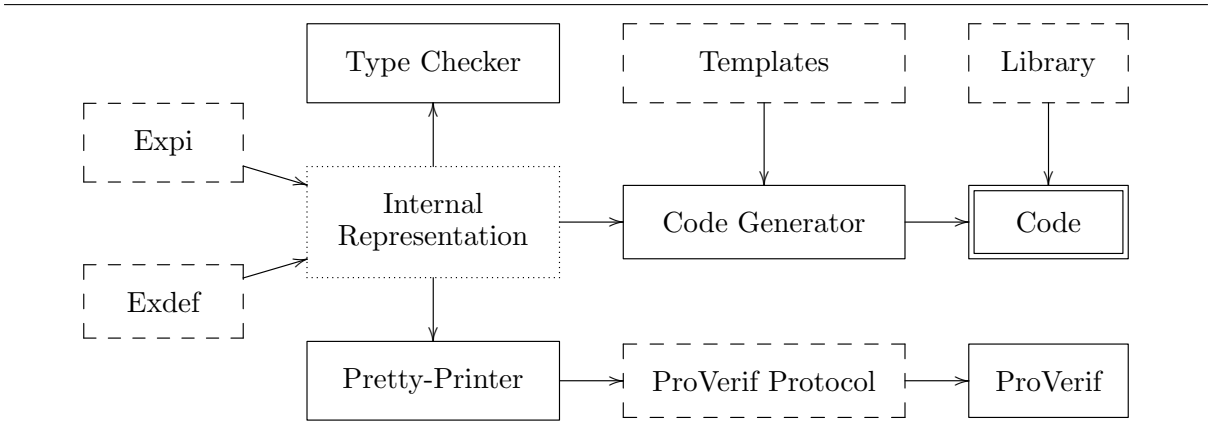

Figure 1: Overall design of Expi2java

2.2 Workflow

In order to generate Java code using Expi2java the user needs to perform several steps:

1. Write an abstract protocol specification in the Extensible Spi Calculus
2. Annotate the abstract specification with the type information
3. Adjust the configuration of all types to match the desired low-level format
4. Automatically generate the code

It is also possible to use an incremental approach and repeat the first three steps several times, each time extending the implementation by another small part of the protocol (e.g., message by message).

Additionally, you might first need to extend Expi2java to support custom data structures and/or cryptographic primitives. Various ways to customize Expi2java are described in detail in Section 6.

3 Input Language

3.1 Calculus

The protocol specifications used by Expi2java are written in a variant of the Spi calculus [AB05, AG99] we call the "Extensible Spi Calculus". The Extensible Spi Calculus uses a subset of the syntax used in ProVerif [Bla08] (a popular automatic cryptographic protocol verifier) with several extensions. The file format of the protocol specifications is called "Expi".

The keywords of the Extensible Spi Calculus are: **data**, **else**, **free**, **fun**, **in**, **let**, **new**, **out**, **private**, **reduc**, **process**, **reduc**. The following keywords are used in ProVerif and are therefore also reserved in the Extensible Spi Calculus for future extensions: **attacker**, **choice**, **equation**, **event**, **if**, **phase**, **query**, **then**, **clauses**, **elimtrue**, **ev**, **evinj**, **mess**, **noninterf**, **not**, **nounif**, **param**, **pred**, **suchthat**, **weaksecret**.

In addition to the Expi files, expi2java uses "External Definitions" files (short Exdef) for the declarations of the constructors and destructors, type definitions and type configurations.

The following keywords are reserved in the Exdef files: **abstract**, **class**, **config**, **data**, **extends**, **fun**, **generative**, **private**, **reduc**, **string**, **type**, **typedef**.

Identifiers are words beginning with a letter (uppercase or lowercase) and contain any number of letters, digits or underscore characters (_). All constructs are case-sensitive. The keywords mentioned before can not be used as identifiers in the corresponding files. The text surrounded by `(*` and `*)` is treated as a comment. Nested comments are not supported.

We provide a syntax highlighting script for Vim that is also suitable for ProVerif input files in Spi format. It can be found in the `vim/syntax/` directory in the Expi2java distribution. To enable syntax highlighting for all `*.expi`, `*.exdef` and `*.pvi` files, copy the file `vim/syntax/spi.vim` into your Vim syntax directory (`$HOME/.vim/syntax/` on UNIX-like systems) and add the following commands into the `$HOME/.vimrc` file:

```
syntax on
autocmd BufRead,BufNewFile *.exdef,*.expi,*.pvi setlocal ft=ocaml syntax=spi
```

Listing 1: Enabling syntax highlighting for Expi files in Vim

3.1.1 Expi Files

An Expi input file consists of a list of declarations or include directives, followed by the keyword **process** and the main process:

$$(\langle \textit{declaration} \rangle \mid \langle \textit{include} \rangle)^* \texttt{ process } \langle \textit{process} \rangle$$

In this grammar X^* means any number of repetitions of X, $[X]$ means X or nothing, $X \mid Y$ means either X or Y and *seq* X is a short form for $(X\texttt{,})^*X$ (a non-empty sequence of $X\texttt{,}\ldots\texttt{,}X$). The parentheses are used for grouping. Text in typewriter style should appear in the file as is. Text enclosed in ⟨ and ⟩ represents non-terminals.

The include directive

```
include "file.exdef"
```

instructs the parser to read additional declarations from an Exdef file `file.exdef`. The Exdef files are discussed in Section 3.1.2. This is an extension to the ProVerif syntax.

An Expi ⟨*declaration*⟩ can be any of the following (note the dot at the end of line):

- [`private`] `free` *seq* (⟨*ident*⟩ `:` ⟨*typeanno*⟩)`.`
  The construct `free` $a_1,\ldots,a_n$ declares the free names $a_1,\ldots,a_n$. The keyword **private** is supported for compatibility with ProVerif. All free names in the processes must be declared as such.

  An extension to the ProVerif syntax is the type annotation "`:` ⟨*typeanno*⟩" that defines the type of the declared free name. The types are described in detail in Section 3.2.

  The free names are treated as protocol parameters in the generated code (see the code template `procs/free` from the templates directory `data/expi2java/javaTemplates/`). They need to be manually initialized by the programmer before the protocol starts and must be deinitialized after the protocol terminates. The value of free variables is not changed during the protocol execution. See Section 6.6 for more information on code templates.

- `let` [⟨*kind*⟩] ⟨*procident*⟩ `=` ⟨*process*⟩`.`
  Defines a process named ⟨*procident*⟩ to be ⟨*process*⟩. Currently, named processes can only be used *once* due to a limitation of the type system.

  An extension to the ProVerif syntax is the optional process kind annotation, where ⟨*kind*⟩ is one of the keywords {`public`, `inline`, `skip`}. The protocol participants are represented as *public* processes (default), *inline* processes represent integral parts of the public processes and processes marked with *skip* are ignored during the code generation. Inline processes can be used to split large public processes into several smaller processes for readability or define helper processes that should not be implemented (e.g., for verification with ProVerif).

The code generated for *inline* processes is inlined into the surrounding code using the code template `procs/call`. *Public* processes are generated using the corresponding process code template `procs/call-public`. Marking a named process with *skip* is equivalent with replacing it by the null process `0` before type checking and code generation.

Terms and Processes

The syntax of the terms and processes is given in Table 1.

Table 1 Grammar of terms and processes

⟨*typedefname*⟩	::=	`$`⟨*ident*⟩
⟨*configname*⟩	::=	`@`⟨*ident*⟩
⟨*typename*⟩	::=	⟨*ident*⟩[`(`⟨*configname*⟩`)`][`<`*seq* ⟨*typename*⟩`>`]
	|	`(`*seq* ⟨*typename*⟩`)`
⟨*typeanno*⟩	::=	⟨*typename*⟩
	|	⟨*typedefname*⟩
⟨*partanno*⟩	::=	⟨*typeanno*⟩
	|	⟨*configname*⟩
⟨*paramanno*⟩	::=	`[`*seq* ⟨*partanno*⟩`]`
	|	`[->` ⟨*partanno*⟩`]`
⟨*term*⟩	::=	⟨*ctorident*⟩[⟨*paramanno*⟩]`(`[*seq* ⟨*term*⟩]`)`
	|	`(`*seq* ⟨*term*⟩`)`
	|	⟨*ident*⟩
⟨*dtorapp*⟩	::=	⟨*dtorident*⟩[⟨*paramanno*⟩]`(`*seq* ⟨*term*⟩`)`
	|	⟨*term*⟩
⟨*pattern*⟩	::=	⟨*ident*⟩
	|	`(`*seq* ⟨*pattern*⟩`)`
⟨*process*⟩	::=	`(` ⟨*process*⟩ `)`
	|	⟨*procident*⟩
	|	`0`
	|	`new` ⟨*ident*⟩ `:` ⟨*typeanno*⟩`;` ⟨*process*⟩
	|	[`!`]`in(`⟨*term*⟩`,` ⟨*pattern*⟩`)`[`;` ⟨*process*⟩]
	|	`out(`⟨*term*⟩`,` ⟨*term*⟩`)`[`;` ⟨*process*⟩]
	|	`let` ⟨*pattern*⟩ `=` ⟨*dtorapp*⟩ `in` ⟨*process*⟩ [`else` ⟨*process*⟩]
	|	⟨*process*⟩ `|` ⟨*process*⟩

The terms in the Extensible Spi Calculus are used to model data. There are three kinds of terms, *identifiers:* ⟨*ident*⟩, *constructor* applications: ⟨*ctorident*⟩[⟨*paramanno*⟩]`(`[*seq* ⟨*term*⟩]`)` and *tuples:* `(`*seq* ⟨*term*⟩`)`. Identifiers represent either *names*, which are used to model constant data, or *variables* that are bound to another terms on runtime. Syntactically, variables are the identifiers ⟨*ident*⟩ inside patterns (see ⟨*pattern*⟩ inside the `in` and `let` processes). Names are

either free (declared using the `free` construct) or restricted to a process using the restriction process (`new` ⟨*ident*⟩).

Constructors are function symbols that are used to build terms. *Destructors* are partial functions that can be applied to terms in the `let` process. An extension to the ProVerif syntax is the optional type annotation ⟨*paramanno*⟩ that defines either the types and/or configuration of all parameters or the return type and/or configuration. Changing the configuration of the return type is often needed when using channels. The types are described in detail in Section 3.2. The constructors and destructors are declared in the Exdef files as described in Section 3.1.2.

Tuples model concatenation of several terms. They are used in the same way as constructors, except that they do not support type annotation.

The *processes* are used to model the behavior of protocol participants and the communication between them [AB05]. The processes can be grouped using parentheses or declared as named processes for convenience in the same way as in ProVerif. Other supported processes are:

- `0`
  The null process is used to stop the execution of a protocol.

  The implementation is generated using the code template `procs/null` from the templates directory `data/expi2java/javaTemplates/`. The default implementation does nothing.

- `new` a `:` ⟨*typeanno*⟩`;` P
  The restriction process generates a fresh name a and then executes the process P.

  An extension to the ProVerif syntax is the type annotation "`:` ⟨*typeanno*⟩" that defines the type of the generated name. The types are described in detail in Section 3.2.

  Unlike the free names, the fresh names are initialized and destroyed automatically during the protocol execution. The implementation of the restriction process is generated using the code template `procs/new` that creates a new instance of the implementation class and uses the *configuration* of the name to set its initial value.

- `in(`c`,`x`)`[`;`P]
  The input process inputs a message from the channel c, binds it to the variable x and then executes the process P. The received data can be accessed in P using the name x.

  The implementation is generated using the code template `procs/in` from the templates directory `data/expi2java/javaTemplates/`. It relies on the implementation class of the channel to receive and decode the message.

  Channels do not only allow for sending and receiving messages on the network, but also model all other types of I/O and even user interaction. See Section 3.1.4, page 13 for an overview over the provided channel implementations.

- `out(`c`,`M`)`[`;`P]
  The output process outputs the term M on the channel c and then executes the process P.

  Note that there is a difference in semantics of the communication channels in the abstract calculus and the channel implementations. In the Spi calculus an output with a continuation process P is *synchronous*, meaning that P is executed after the message M is *received*

by another participant. The channel implementations use another semantics, where synchronous means that the continuation process is executed as soon as the message is *sent* on the channel (blocking I/O). *Asynchronous* communication in Spi means execution of several output processes in parallel (implemented using threads), which means that the messages are received in arbitrary order. At the moment we do not support the usual concept of non-blocking I/O, where the next process of the sender may start during the message is still transmitting and the receiver polls the input channel.

The implementation is generated using the code template `procs/out`. The identifier terms are used as is. Constructor applications are implemented using the code template `procs/constructor` that applies the corresponding method of the implementation class to all parameters. The resulting term is serialized and sent over the channel according to the implementation of the channel type. See Section 3.1.4, page 13 for an overview over the provided channel implementations.

- `let` x `=` g`(`*seq* M_i`)` `in` P [`else` Q]
  The let process applies the destructor (or constructor) g to the arguments M_i and, in case the destructor succeeds, binds its result to the name x and then executes the process P. If the destructor fails and there is an else branch available, process Q is executed, otherwise the execution is terminated.

  This is the only place where a destructor can be used. Nested destructors are not supported.

  The implementation of the let process is generated using the code templates `procs/let`, `procs/destructor` for destructor application and `procs/constructor` for constructor application and nested terms. The destructor application is currently implemented as a method of the corresponding implementation class. It throws an exception on errors that is used to control the further control flow. See Section 6.4 for more information on the implementation of constructors and destructors. See Section 6.6 for more information on code templates.

- P | Q
  The parallel composition executes the processes P and Q concurrently.

  In most cases, parallel composition is used together with named processes to model the protocol participants. By convention, every participant should be modelled as a named process, which should be in a parallel composition with all other participant processes. The implementation can select one of the participant processes on request or run them all in parallel for debugging purposes. The behavior of named processes in a parallel composition can be adapted using the process kind annotations (see Section 3.1.1).

  The implementation generated using the code template `procs/parallel` uses threads to model the parallel execution. It starts both processes and waits until both of them finish before terminating. Threads are generated using the code templates `procs/thread-call` and `Thread_%PROC_NAME%_%PROC_ID%.java`.

- `!in(`c`,`x`);`P
  The replication models an unbounded number of processes executed in parallel, i.e., in this case, `!in(`c`,`x`);`P = `in(`c`,`x`);`P | ... | `in(`c`,`x`);`P | ....

A specific characteristic of the Extensible Spi Calculus is that a replication can only be followed by the input process, as in [FGM07, BHM08], since this is the most common way to use replication in a protocol specification.

The default implementation of the replication `!in(`c`,`x`);`P runs the input process in an endless loop and starts the process P in a new thread every time a message is received on the channel c. Neither the thread running the main loop, nor other threads running other copies of process P wait for each other. This implementation is generated using the code templates `procs/replicated` and `procs/thread`.

3.1.2 Exdef Files

An Exdef input file consists of a list of declarations and include directives:

$$(\langle \textit{exdef-declaration} \rangle \mid \langle \textit{include} \rangle)^*$$

The include directive

```
include "file.exdef"
```

instructs the parser to recursively read declarations from another Exdef file `file.exdef`.

The syntax of the Exdef declarations is given in the Table 2 using the same convention as in Section 3.1.

An Exdef $\langle$*exdef-declaration*$\rangle$ can be any of the following (note the dot at the end of line):

- `config` [`abstract`] A [`extends` B]`(key = "value", type Type(class = "MyClass")).`
  Declares a configuration named A that uses the implementation class `MyClass` for the type `Type` and sets the key named `key` to the string value `value`. If the keyword **abstract** is present, the implementation class key becomes optional. If the `extends` B part is present, the configuration A extends an existing configuration B with additional keys (new keys with the same name take precedence). See Section 3.1.4 below for more information about configurations.

- `type` [`generative`] $Type$[`@`$Config$][`<W, X+, Y-, Z0>`]`.`
  Defines a new type named $Type$. If the keyword **generative** is present, a fresh name of this type can be generated in a restriction process. If the default configuration $Config$ is specified, it is used automatically if the type is used without an explicit configuration. The list of parameters can be used to specify the type parameters together with their variance (see Section 6.3 for more information about subtyping and variance). In our case, the type is parametrized with four parameters and is invariant (by default) in the first, covariant in the second, contravariant in the third and invariant in the fourth parameter with respect to subtyping. See Section 3.2 for more information about the use of types.

- `typedef` $\$Short$ `=` $AType$`@`$WithConfiguration$`<`And`,` $Many$`<`$\$Parameters$`>>.`
  Defines a typedef called $\$Short$ for a long nested type. The typedef name $\$Short$ will be replaced by the right hand side of the typedef every time it is used in a type.

- `fun` f `: [`$T_1, \ldots, T_k$`].` `(`$U_1, \ldots, U_n$`) ->` V`.`
  Declares a constructor f with n arguments and k type parameters $T_1, \ldots, T_k$. The argument types are $U_1, \ldots, U_n$ and the return type is V. All type parameters occurring in the argument and return types should be listed in the list of type parameters.

Table 2 Grammar of the Exdef declarations

⟨*sparams*⟩	::=	`class = "`⟨*ident*⟩`"[,` *seq* (⟨*ident*⟩ `= "`⟨*string*⟩`")]`
⟨*section*⟩	::=	`type` ⟨*ident*⟩`(seq` ⟨*sparams*⟩`)`
⟨*params*⟩	::=	`[bundle = "`⟨*string*⟩`",][seq (`⟨*ident*⟩ `= "`⟨*string*⟩`")][,` *seq* ⟨*section*⟩`]`
⟨*config*⟩	::=	`config [abstract]` ⟨*ident*⟩ `[extends` ⟨*ident*⟩`](`⟨*params*⟩`).`
⟨*var*⟩	::=	⟨*uppercaseletter*⟩
⟨*typevar*⟩	::=	⟨*var*⟩ | ⟨*var*⟩`+` | ⟨*var*⟩`-` | ⟨*var*⟩`0`
⟨*typedecl*⟩	::=	`type [generative]` ⟨*ident*⟩`[@`⟨*ident*⟩`][<seq` ⟨*typevar*⟩`>].`
⟨*typedefname*⟩	::=	`$`⟨*ident*⟩
⟨*typename*⟩	::=	⟨*ident*⟩`[@`⟨*ident*⟩`][<seq` ⟨*typename*⟩`>]`
	|	`(seq` ⟨*typename*⟩`)`
	|	⟨*typedefname*⟩
⟨*typedef*⟩	::=	`typedef` ⟨*typedefname*⟩ `=` ⟨*typename*⟩`.`
⟨*tnamesimple*⟩	::=	⟨*ident*⟩`[<seq` ⟨*tnamesimple*⟩`>]`
	|	⟨*var*⟩
⟨*funtype*⟩	::=	`[[seq` ⟨*var*⟩`].] [([seq` ⟨*tnamesimple*⟩`]) ->]` ⟨*tnamesimple*⟩
⟨*ctor*⟩	::=	`fun` ⟨*ident*⟩ `:` ⟨*funtype*⟩`.`
⟨*dtor*⟩	::=	`reduc (`⟨*ident*⟩`(seq` ⟨*term*⟩`) =` ⟨*term*⟩`;)`* ⟨*ident*⟩`(seq` ⟨*term*⟩`) =` ⟨*term*⟩ `:` ⟨*funtype*⟩`.`

- `reduc` g`(`$M_1, \ldots, M_n$`) =` N`;`
  ...
  g`(`$M'_1, \ldots, M'_n$`) =` N' `: [`$T_1, \ldots, T_k$`].` `(`$U_1, \ldots, U_n$`) ->` V`.`
  Declares a destructor g with n arguments, k type parameters $T_1, \ldots, T_k$ and multiple reduction rules. Every reduction rule defines how the terms are transformed in a destructor application. The argument types of all reduction rules should be $U_1, \ldots, U_n$ and the type of the result terms $N, \ldots, N'$ should be V.

3.1.3 Constructors and Destructors

We provide a number of default constructors and destructors that model commonly used cryptographic primitives and data structures (see file `testData/exdef/default.exdef`). Constructors are function symbols that are used to build terms. They can e.g., represent encrypted messages, hashes or boolean values. Destructors are partial functions that can be applied to terms in the `let` process. Every destructor is defined using one or more reduction rules that define what happens to the arguments. For example, the decryption destructor `dec` reflects the main property of symmetric encryption schemes, namely that decrypting an encrypted message with the correct key results in the plain-text message.

The list of default constructors and destructors consists of (please ignore the types for now):

```
fun enc   : [T]. (T, SymKey<T>) -> SymEnc<T>.
fun enca  : [T]. (T, PubKey<T>) -> PubEnc<T>.
fun pk    : [T]. (KeyPair<T>) -> PubKey<T>.
fun sk    : [T]. (KeyPair<T>) -> PrivKey<T>.
fun sign  : [T]. (T, SigKey<T>) -> Signed<T>.
fun vk    : [T]. (SigKey<T>) -> VerKey<T>.
fun h     : [T]. (T) -> Hash<T>.
fun true  : Bool.
fun false : Bool.
fun pair  : [T, U]. (T, U) -> Pair<T, U>.
fun succ  : (Int) -> Int.
fun zero  : Int.
fun iv    : [T]. (SymKey<T>, Int) -> SymKey<T>.
fun hmac  : [T]. (T, SymKey<T>) -> Hash<T>.

reduc dec(enc[T](x, y), y) = x          : [T]. (SymEnc<T>, SymKey<T>) -> T.
reduc deca(enca[T](x, pk[T](y)), sk[T](y)) = x
                                        : [T]. (PubEnc<T>, PrivKey<T>) -> T.
reduc msg(sign[T](x, y)) = x            : [T]. (Signed<T>) -> T.
reduc ver(sign[T](x, k), vk[T](k)) = x  : [T]. (Signed<T>, VerKey<T>) -> T.
reduc eq(x, x) = true                   : [T]. (T, T) -> Bool.
reduc first(pair[T, U](x, y)) = x       : [T, U]. (Pair<T, U>) -> T.
reduc second(pair[T, U](x, y)) = y      : [T, U]. (Pair<T, U>) -> U.
reduc pre(succ(x)) = x;
      pre(zero)    = zero()             : (Int) -> Int.
reduc id(x) = x                         : [T]. (T) -> T.
reduc accept(c) = c                     : [T]. (Channel<T>) -> Channel<T>.
```

Listing 2: Default constructors and destructors

Symmetric encryption is modelled using enc, dec and iv. The initialization vector (if it is used) is assumed to be stored together with the key and can be changed using the iv constructor. Asymmetric encryption is modelled using enca, deca, sk and pk. The sk and pk constructors are used to extract the private and public keys from the key pair respectively.

Digital signatures are modeled using sign, ver, msg and vk. The difference between ver and msg is that ver fails is the signature is not valid and msg always extracts the message. Note that signatures use a different key type than asymmetric encryptions.

Hashes are modelled using constructors h (unkeyed hash functions or message digests) and hmac (keyed hash functions or MACs).

Two constructors true and false model the corresponding boolean values. Simple integer arithmetic can be modelled using zero, succ and pre. The constructor zero represents the integer constant 0. Every integer can be incremented using succ and decremented using pre.

The constructor pair can be used to combine several data fields. Both pair elements can be accessed using the first and second destructors. Current implementation of pairs simply concatenates both elements, this means that the terms pair(pair(a, b), c) and pair(a, pair(b, c)) result in the same binary representation in the generated code (e.g., when sent on the network), although they are semantically different in the Extensible Spi Calculus.

A more convenient way to model concatenation is to use *tuples*. Tuples are a syntactic sugar for nested pairs, e.g., the tuple (a, b, c, d) is equivalent to pair(a, pair(b, pair(c, d))). Assigning to a tuple in the **let** process or receiving it in the **in** process can be used to split the

tuple into its components. For example, **let** (a, b) = dec(e, k) **in** P decrypts the message e using the key k and splits the resulting tuple, creating two names a and b. This example is equivalent to **let** tmp = dec(e, k) **in** **let** a = first (tmp) **in** **let** b = second(tmp) **in** P. Analogously, **in**(c, (a, b)); P is equivalent to **in**(c, tmp); **let** a = first (tmp) **in** **let** b = second(tmp) **in** P.

The destructor eq can be used to test for equality of two terms. Note that in the Extensible Spi Calculus, equality means structural equivalence only and the implementation additionally compares the values of all contained names at runtime. This destructor never returns false, it fails (and aborts the protocol execution in case there is no else branch) if the terms given as arguments are not equal.

The identity destructor id returns its argument and binds it to a new variable in the **let** process, which can be used to change the configuration of some name or variable.

The accept() destructor is used together with channels (especially TCP/IP channels). It models the behavior of the server, i.e., listens on the corresponding port until some client connects to it, then establishes the connection, sets up and returns a communication channel. The configuration of the used channel needs to be changed after application of accept() as described in Listing 5.

The constructors and destructors can have parametric type. In this case, all used type parameters are listed in the type parameter list in the declaration. The type checker tries to infer the types of all parameters before type checking, however it might fail to infer the exact type in some cases (e.g., it is impossible to infer a non-default configuration of the return type). In such cases the type and/or configuration of every type parameter must be specified explicitly on every application of a constructor or destructor. This is done using a type annotation of the form [*seq* ⟨*partanno*⟩]. The type parameters are substituted with the types from the list in the order of occurrence in the declaration of the corresponding constructor or destructor. If a partial annotation is used (only the configuration name), the types of parameters are first inferred and then their configurations are replaced by the provided ones. Please note that all parametric constructors inside destructor declarations must always be annotated with the used types or type parameters.

In some cases it is more important to annotate the result of a constructor or destructor application using the alternative form of the type annotation: [-> ⟨*partanno*⟩]. For example, one can select the used hash algorithm of a hash function as follows: h[-> @SHA1](msg). The types of all parameters are inferred in this case. It is not allowed to specify both types of type parameters and the return type at the same time.

3.1.4 Configurations

The specifications of real-world protocols contain a lot of information that cannot be directly expressed in the Extensible Spi Calculus, but still should be taken into account during the code generation. For example, there are often exact rules of how the messages should look like on the bit-string level, which data types should be used, which size they should have etc. In Expi2java this information must be specified for every term using *configurations*.

The configurations are simple key-value mappings. They are defined in the Exdef files using the syntax from Table 2. There is also a way to extend the configurations, similar to the inheritance

in object-oriented programming. This allows the user to define a configuration that only differ in several keys from another one in a convenient way. The keys "class" and "bundle" are special.

Type Sections The configurations usually provide implementation-specific settings that only make sense for a couple of types. In Expi2java the user must explicitly list which types can use a configuration by specifying *type sections* within every configuration. Each type section must provide the name of the implementation class in the special "class" key. Other settings, like the employed algorithm, padding or key length, can either be specified for all type sections or only for specific implementation class. The implementation classes are given these settings as a mapping and are free to use them however they like.

Bundles In most cases, a configuration has only one type section, however it is often important to use the same configuration with different types. For example, we need different low-level settings for the types $\mathsf{SymKey}_{\mathsf{AES}}\langle T\rangle$, and $\mathsf{SymEnc}_{\mathsf{AES}}\langle T\rangle$, but they both should be part of the same configuration called "AES". In this way the type checker can detect errors, such as decrypting a DES encrypted message with an AES key.

We call configurations that can be used with several types *bundles*. As an example, the AES bundle is shown in Listing 3:

```
(* AES bundle *)
config AES(
    algorithm   = "AES", (* common settings for all types *)
    keylength   = "256",
    provider    = "SunJCE",
    type SymEnc(
        class   = "SymEnc", (* implementation class (mandatory) *)
        mode    = "CTR",    (* specific settings for SymEnc *)
        padding = "PKCS5Padding"
    ),
    type SymKey(
        class   = "SymKey",
        iv      = "1234567887654321"
    ),
    type Int( (* initialization vectors *)
        class   = "BigNonce",
        size    = "16" (* nonce size for IV *)
    )
).
```

Listing 3: AES bundle

It is also possible to bundle several different configurations into one virtual bundle using a special key "bundle". To enforce the use of bundles we require that all terms used in the same constructor or destructor should use configurations that belong to the same bundle. The terms that have a type variable (like T or U) as the type in the constructor or destructor declaration are excluded from this rule.

```
(* common RSA values *)
config abstract RSA_(
    algorithm           = "RSA",
    keylength           = "512",
    provider            = "SunJCE",
```

```
    mode                = "ECB",
    padding             = "PKCS1Padding"
).

(* RSA encryption bundle *)
config RSA extends RSA_(
    bundle              = "RSA Encryption",
    type PubEnc(class   = "PubEnc"),
    type PubKey(class   = "PubKey"),
    type PrivKey(class  = "PrivKey"),
    type KeyPair(class  = "KeyPair")
).
config RSACertificate extends RSA_(
    bundle              = "RSA Encryption",
    type PubKey(
        class           = "TLSCertificate",
        provider        = "BC", (* bouncy castle *)
        cert_type       = "X509" (* for TLS *)
    )
).
```

Listing 4: RSA bundle

The RSA bundle shown in Listing 4 uses the bundle key to define the configuration RSACertificate used in TLS. This configuration can only be used with public keys, but is still a part of the main encryption configuration RSA and thus can be used in enca constructor as a normal public key.

Default Configurations We provide default configurations for the most commonly used cryptographic primitives and data types (see file `testData/exdef/configs.exdef`). These configurations are designed to be used together with the default types listed in Section 3.2 on page 17. The list of default configurations currently consist of:

AES Encryption Bundle
: AES
  The AES bundle represents symmetric encryptions. The AES configuration can be used to represent AES-encrypted data, the corresponding keys and the initialization vectors. The implementation supports various configuration options, such as length of the key, block chaining mode, used cryptographic provider etc.

DES Encryption Bundle
: DES
  The DES bundle is similar to the AES bundle. In fact, all symmetric encryptions are implemented using the same classes and have the same configuration options

RSA Encryption Bundle
: RSA, RSACertificate
  The RSA encryption bundle represents asymmetric encryptions. The RSA configuration can be used to represent RSA-encrypted data, public keys, secret keys and key pairs. Additionally, RSACertificate can be used to represent X509 certificates.

RSA Signature Bundle
: RSASign

The RSA signature bundle represents asymmetric signatures. The RSASign configuration can be used to represent signed messages, signing keys and verification keys. The implementation classes of signatures can also handle many other signature schemes, as long as the used cryptographic providers supports them.

Hashes

SHA1, MD5

The hash configurations define the used cryptographic hash function. The algorithm and the hash length can be changed using the corresponding keys. The provided configurations are preconfigured with the two popular algorithms, SHA1 and MD5.

Channels

TcpIpChCfg, ConsoleChCfg, FileChCfg, MemoryChCfg, KeyStoreChCfg, ReturnChCfg

The channels are more complex and can be used to implement very different behaviors.

The communication over the network can be done using the **TCP/IP channel**. A specific characteristic of the TCP/IP channel is the client-server separation. The communication can only take place if one of the participants acts as a server, meaning that it should listen on a given port for connections, while clients try to connect to the server and abort on failure. TCP/IP channels are created as client channels, they can be reconfigured into a server channel using the accept() destructor. Both configurations should differ in the value of the "variable" key as discussed in the "Channel Splitting and Merging" paragraph of Section 3.1.4, page 15.

Console channel is used to model user interaction. All messages sent on a console channel are printed on the screen, receiving corresponds to a user prompt. The type of the message must be explicitly supported in order to produce the expected results. At the moment, integers, strings and boolean values can be read from, and any type can be written to a console channel.

File channel allows for reading and writing files in the obvious way. The messages are converted from/to the raw data format using the serialization/deserialization methods of the corresponding implementation class. The same serialization format is used with all other channels except for the console channels.

Memory channel is used for fast communication within several threads of the same process. The messages are sent in a synchronous way, both sender and receiver threads block until the data is transmitted.

Key store channel represents an interface to the Java key store mechanism used to store the keys in a secure way. A key can be stored under some name by sending the pair (name, key) on a key store channel. A previously stored key can be retrieved by sending the name and then receiving the stored key on the same channel.

Return channel is used to store the results created during a protocol execution and return them to the application after the protocol terminates. A channel that uses this configuration should be declared as free, otherwise it will be deinitialized after the end of the protocol. The values can be stored by sending them on the return channel. They are stored in a variable-value map that can be read using the corresponding methods.

Strings

IdentifierCfg, ASCIIStringCfg, UTF8StringCfg

The string configurations differ in used string representations. The ASCIIStringCfg stores the string in ASCII encoding and is useful for retrieving HTML or plain-text data. All other configurations use UTF-8 encoding. IdentifierCfg is used to represent participant names. UTF8StringCfg represents the Java String and uses the same serialization methods. This configuration can be used in the **new** process that generates new variables. The value of these variables is set using the configuration key "message".

Nonces

BigNonceCfg, NonceCfg

The nonces are integers that are initialized to a random value using the **new** process. NonceCfg uses an ordinary 64 bit integer to store the data and supports incrementing and decrementing, BigNonceCfg uses a byte array of arbitrary length.

Integers

IntCfg, UInt8, UInt16, UInt24, UInt32, UInt40, UInt48, UInt64

In Expi2java integers can have any length from 8 to 64 bit in 8 bit steps. We provide a configuration with configurable length and several configurations with predefined length. Integers are given their value in a similar way as strings, using the key "value" in the configuration.

BoolCfg

The boolean values are implemented as one byte with the value 0 or 1.

TimestampCfg

The time stamp stores the current time on creation in a **new** process. It uses the UNIX time format for serialization.

RawDataCfg

We provide a special data type for passing any data as-is. The raw data is handled in a similar way as byte arrays, except that it always consumes as much data as possible.

PairCfg

The implementation of pairs concatenates the binary representation of both components.

Channel Splitting and Merging In Expi2java the user very often needs to split a channel in order to type check the protocol. For instance, consider a protocol, where Alice sends a nonce to Bob in the first message and Bob answers with a symmetric encryption of this nonce using a shared key. Since the type of the first message is Nonce, the channel must have the type Channel<Nonce>. However, the second message has another type, SymEnc<Nonce>, so the channel must have the type Channel<SymEnc<Nonce>>, which is incompatible with Channel<Nonce>. In order to type check such protocols the channel must be *split* into several channels, one for every message.

These split channels need to be *merged* together to "logical" channels in the generated code, but often in a different way as they would be represented in the untyped Extensible Spi Calculus. The implementation of a TCP/IP channel, for instance, needs to share a reference to a server socket among all Java variables that represent the channel names used in the participant process that acts as a server, and another reference to a client socket among the Java variables that represent all other parts of the same channel.

As an example, consider a three-message protocol with two participants, where all messages have different types (see Section 3.2 for more information on types). As illustrated in Figure 2, in the untyped Spi calculus we can use one channel name `c` to send all three messages. In the typed Extensible Spi Calculus we must use three channel names `c1`, `c2`, `c3`, because the messages have different types. Finally, in the generated code we need to use two Java channels due to the server-client nature of TCP/IP.

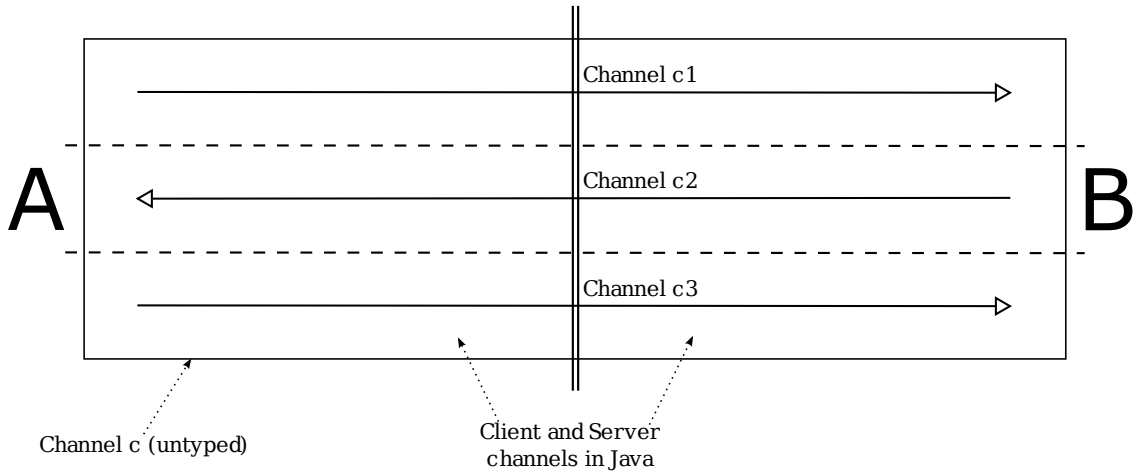

Figure 2: Channel splitting in Perrig-Song protocol

We have implemented channel merging using special implementation classes, called *aliases*. An alias controls the mapping between the Expi variables in the protocol specifications and the Java variables in the generated code. The user can define virtual variables in the generated code using the configuration key "variable". The aliases create one instance of the aliased class for every virtual variable. Every instance of an alias in the generated code that belongs to the same virtual variable forwards all method calls to the corresponding instance of the aliased class, effectively acting as one Java variable.

In practice, we need two configurations for every communication channel, one used on creation and another after accepting:

```
new c1 : Channel@ClientCfg<$Msg1>; (* client channel for message 1 *)
new c2 : Channel@ClientCfg<$Msg2>; (* client channel for message 2 *)
...
let c_server1 = accept[->@ServerCfg](c1) in
let c_server2 = accept[->@ServerCfg](c2) in
    ...
```

Listing 5: Client-server channels setup

To model a common communication channel represented by two channel names `c1` and `c2` from Listing 5 we use different configurations for the client and server sides. The configurations `ClientCfg` and `ServerCfg` are equal up to the value of the configuration key "variable". The implementation will assure that both channels `c1` and `c2` will use the same socket on the client side, and the channels `c_server1` and `c_server2` will use the same socket on the server side. The connection is also made only once, on the first call to the `accept()` destructor, later calls are ignored.

3.2 Types

Every term in the Extensible Spi Calculus must be given a type corresponding to its data type, such as integer, string, symmetric encryption, pair etc. This information is also used to express the intended use of constructors and destructors. The user can select the correct implementation class for each term and configure the implementation classes according to the specification with the help of configurations. The configurations are an integral part of types. A type checker statically enforces the proper usage of configurations and the corresponding constructors and destructors.

Currently, the type of every free name or name created using the restriction process must be specified by hand using the type annotations. In most cases the type checker can infer the parameter types of parametric constructors and destructors, if it fails, they must also be annotated with the type of all parameters by hand.

There are only three built-in types, Top, Channel$\langle T \rangle$ and a special tuple type:

```
type              Top.
type generative   Channel<T0>.
type              (A+, B+, ..., Z+). (* special type with variable length *)
```

Listing 6: Built-in types

The type Top corresponds to the type Object in Java. Any term having type T can also be seen as having type Top. This allows us to express for instance keys that can encrypt messages of any type or channels that can handle messages of any type. See also the paragraph on subtyping in Section 6.3.

The tuple type is given to *tuples*. Tuples model concatenation of 2 to 26 values with different types. The tuple type is special in the sense that the tuple type of correct length is taken for each tuple automatically. Its configuration can not be changed.

Default Types Expi2java provides types for the most commonly used cryptographic primitives and basic data structures:[1]

SymEnc<T+>
: Symmetric encryption of a message of type T. In cases where the exact type of the message is not important, SymEnc$\langle$Top$\rangle$ can be used.

SymKey<T−> (generative)
: Symmetric key for messages of type T. SymKey$\langle$Top$\rangle$ represents an universal key that can be used with messages of any type.

PubKey<T−>
: Public encryption key for messages of type T. PubKey$\langle$Top$\rangle$ can be used with messages of any type.

PrivKey<T−>
: Private (secret) decryption key for messages of type T. PrivKey$\langle$Top$\rangle$ can be used with messages of any type.

[1] Additional types can be added as explained in Section 6.3.

KeyPair<T−> (generative)
: Pair of the corresponding private and public keys for messages of type T. This type is used to generate new keys in a **new** process. KeyPair⟨Top⟩ can be used with messages of any type.

PubEnc<T+>
: Asymmetric encryption of a message of type T. In cases where the exact type of the message is not important, PubEnc⟨Top⟩ can be used.

SigKey<T−> (generative)
: Signing key for messages of type T. In many asymmetric encryption schemes the secret key is used as the signing key. SigKey⟨Top⟩ can be used to sign messages of any type.

VerKey<T−>
: Verification key for messages of type T. In many asymmetric encryption schemes the public key is used as the verification key. VerKey⟨Top⟩ can be used to verify messages of any type.

Signed<T+>
: Signed message of type T. The implementation assumes that the signed messages contain not only the signature, but also the message. In cases where the exact type of the message is not important, Signed⟨Top⟩ can be used.

Hash<T+>
: The result of applying a hash function to a message of type T. The hash function may be keyed (i.e., HMAC) or unkeyed (i.e., SHA-1). In cases where the exact type of the message is not important, Hash⟨Top⟩ can be used.

Bool
: Boolean value. This type uses a default configuration[2] BoolCfg.

Int (generative)
: Represents signed or unsigned integers. This type can also be configured as a nonce or an UNIX time stamp.

ExpiString
: Represents string messages for the user, messages received from the network or names of protocol participants. This type uses a default configuration UTF8StringCfg.

Data
: Represents a data type for passing arbitrary data without modifications, similar to a byte array. This type uses a default configuration RawDataCfg.

Pair<X+, Y+>
: A pair of two arbitrary types. This type uses a default configuration PairCfg.

Only some of the types can be used in a restriction process. We call these types *generative*. In the default set of types we allow to generate fresh names of type Int, SymKey⟨T^-⟩, PrivKey⟨T^-⟩, SigKey⟨T^-⟩ and ExpiString. Generating fresh integers and strings initializes them with the value set in the configuration. Fresh keys correspond to random session keys. Generating a fresh channel opens a new communication channel.

Every constructor or destructor has a type. The default constructors and destructors have the following types:

[2]The default configuration is used automatically if no explicit configuration is given.

```
fun enc   : [T]. (T, SymKey<T>) -> SymEnc<T>.
fun enca  : [T]. (T, PubKey<T>) -> PubEnc<T>.
fun pk    : [T]. (KeyPair<T>) -> PubKey<T>.
fun sk    : [T]. (KeyPair<T>) -> PrivKey<T>.
fun sign  : [T]. (T, SigKey<T>) -> Signed<T>.
fun vk    : [T]. (SigKey<T>) -> VerKey<T>.
fun h     : [T]. (T) -> Hash<T>.
fun true  : Bool.
fun false : Bool.
fun pair  : [T, U]. (T, U) -> Pair<T, U>.
fun succ  : (Int) -> Int.
fun zero  : Int.
fun iv    : [T]. (SymKey<T>, Int) -> SymKey<T>.
fun hmac  : [T]. (T, SymKey<T>) -> Hash<T>.

reduc dec(enc[T](x, y), y) = x          : [T]. (SymEnc<T>, SymKey<T>) -> T.
reduc deca(enca[T](x, pk[T](y)), sk[T](y)) = x
                                        : [T]. (PubEnc<T>, PrivKey<T>) -> T.
reduc msg(sign[T](x, y)) = x            : [T]. (Signed<T>) -> T.
reduc ver(sign[T](x, k), vk[T](k)) = x : [T]. (Signed<T>, VerKey<T>) -> T.
reduc eq(x, x) = true                   : [T]. (T, T) -> Bool.
reduc first(pair[T, U](x, y)) = x       : [T, U]. (Pair<T, U>) -> T.
reduc second(pair[T, U](x, y)) = y      : [T, U]. (Pair<T, U>) -> U.
reduc pre(succ(x)) = x;
      pre(zero)    = zero()             : (Int) -> Int.
reduc id(x) = x                         : [T]. (T) -> T.
reduc accept(c) = c                     : [T]. (Channel<T>) -> Channel<T>.
```

Listing 7: Default constructors and destructors with types

Type Definitions A useful feature of Expi2java are the *type definitions* (or short, *typedefs*). The names of typedefs always start with a $ sign to make them visually distinguishable from the types. Unlike types, the typedefs can neither be configured, nor have type parameters (but the mapped types can). The type definitions are pure syntactic sugar, they are textually replaced with the types they are mapped to on parsing. The use of the typedefs can significantly shorten the size of the type annotations.

We provide a list of default typedefs for some often used combinations of a type and the corresponding configuration:

```
typedef $Identifier = ExpiString@IdentifierCfg.
typedef $Nonce = Int@NonceCfg.
typedef $Timestamp = Int@TimestampCfg.
typedef $UInt8  = Int@UInt8.
typedef $UInt16 = Int@UInt16.
typedef $UInt24 = Int@UInt24.
typedef $UInt32 = Int@UInt32.
typedef $UInt48 = Int@UInt48.
typedef $UInt64 = Int@UInt64.
typedef $ReturnCh = Channel@ReturnChCfg<Top>.
typedef $STDOUT = Channel@ConsoleChCfg<Top>.
typedef $STDIN = Channel@ConsoleChCfg<ExpiString>.
```

Listing 8: Default typedefs

4 Command Line Options

The `bin/` directory contains two JAR files, `eXpi2Java.jar` and `eXpiLibrary.jar`

eXpi2Java.jar
: Is the main program, it needs `jalopy.jar`, `log4j.jar` and `bcprov-jdk16-141.jar` from the `lib/` directory.

eXpiLibrary.jar
: Is the runtime library containing the implementations of the cryptographic primitives and data types used by the generated code.

Linux and MacOS X users can also use the provided scripts, which relieve the user from the need to set the CLASSPATH correctly (run them without parameters for short usage help):

expi2java	Encapsulates the main program.
parseExdef	Preselects parameter for parsing exdef files
parseExpi	Preselects parameter for parsing expi files
typeCheckExpi	Preselects parameter for running type-checker
generateAllTests.sh	Generates Java implementation of several examples (no help screen available)
generateTlsChannel.sh	Generates Java implementation of the TLS channel in the library (no help screen available)
run-protocol.sh	Runs one of the generated protocol implementations

The implementation of cryptographic primitives rely on Java cryptographic providers. The default configurations in `configs.exdef` use SunJCE, SUN and BC (Bouncy Castle[3]) providers. The Bouncy Castle provider needs an additional file `lib/bcprov-jdk16-141.jar` that is provided together with expi2java.

Expi2Java can perform 4 basic actions:

--print-exdef
: Pretty-print an Exdef file. Optional switches in this mode are: --debug

--print-expi
: Pretty-print an Expi file. Optional switches in this mode are: --proverif-format, --debug

--type-check
: Type check a protocol given as an Expi file. Optional switches in this mode are: --check-config, --verbose, --debug

--generate
: Generate Java implementation of the protocol given as an Expi file. Mandatory switches in this mode are: --package, --templates, --output. Optional switches are: --name, --application-name, --beautify, --verbose, --debug

[3]The Legion of the Bouncy Castle: http://www.bouncycastle.org/

4.1 Options

All currently supported options are listed below. Every option has a short and a long form. Short options can not be merged.

-e filename.exdef, --print-exdef filename.exdef
: Pretty-print given Exdef file

-x filename.expi, --print-expi filename.expi
: Pretty-print given Expi file

-f, --proverif-format
: Pretty-print in ProVerif-compatible format. Can only be used with -x (optional)

-t filename.expi, --type-check filename.expi
: Type check a protocol given as an Expi file

-c, --check-config
: Check if an Expi protocol is configured for code generation (optional). This checks that every term has a type that uses a non-abstract configuration (i.e., with the "class" key set to a suitable value)

-g filename.expi, --generate filename.expi
: Generate Java implementation of the protocol given as an Expi file

-p full.package.name, --package full.package.name
: Use this package name in the generated code

-m directory/, --templates directory/
: Path to the code templates used in code generation

-o directory/, --output directory/
: Path where all generated files go to. The directory must already exist

-n name, --name name
: Name of this protocol. The name of the input file is used if not set. (optional)

-a appname, --application-name appname
: Name of the generated application class. "Application" is used if not set. (optional)

-b, --beautify
: Format the generated Java code for better readability (warning, slows down the code generation for large protocols). (optional)

-d, --debug
: Enable debug mode (optional). Adds internal unique IDs to the names of all variables, processes, constructors and destructors

-v, --verbose
: Be verbose. Prints the protocol after each step

-s, --subtype-test
: Run a number of internal tests, print results. Only useful for developers

-V, --version
: Print version information and exit

-h, --help
: Print the help screen and exit

Notes

The generated files in the output directory are overwritten if they already exist. If the same parameter is specified multiple times, only the last parameter is used.

--print-exdef, --print-expi, --type-check, --generate, --subtype-test, --version, --help
: Mutually exclusive operations, one of them must be present

--generate
: Implies --type-check and --check-config
Needs --package, --templates and --output

--check-config
: Only has effect if operation is --type-check

--beautify, --name, --application-name
: Only have effect if operation is --generate

--proverif-format
: Only has effect if operation is --print-expi

5 Example: "Hello World" Protocol

We will show the basics of code generation in Expi2java on a simple "Hello World" protocol defined in Figure 3. For the sake of simplicity we will use default configurations (see Section 3.1.4, page 13) for all types.

Figure 3: "Hello World" protocol

5.1 Formal Specification

The "Hello World" protocol consists of one text message sent over a common communication channel. In order to model it in the Extensible Spi Calculus, we need to create a channel, let one participant create a message and send it on the channel and another participant receive and display it.

First, we create two new files, a new specification file `hello-world.expi` and a new configuration file `hello-world.exdef` and put them into `testData/expi/` directory. We include the

configuration file `hello-world.exdef` into the specification file using the include directive. To be able to use the provided default types (see Section 3.2, page 17) and default constructors and destructors from Listing 7 we also include the file `default.exdef`.

We add two processes named pA and pB, and let them be executed in parallel to model the interaction of two participants:

```
(*
 * A simple "Hello World" protocol
 *)

include "../exdef/default.exdef"
include "hello-world.exdef"

let pA =
    0.

let pB =
    0.

process
    (pB | pA)
```

Listing 9: Simple setup: `hello-world.expi`

The channel and the message we will send are modelled as names. We can generate names using the restriction process **new**. Every name has a type and a corresponding configuration that define the implementation class used in the generated code. The initial value of the name can also be controlled through the configuration.

We use thw type Channel together with the configuration TcpIpChCfg_ to model a TCP/IP channel. We adapt this configuration for our needs by defining two new configurations named ChanA and ChanB, both based on TcpIpChCfg_. We need two configurations with different values of the "variable" key to separate the server and client sides of the TCP/IP channel:

```
(* Configuration of the "Hello World" protocol *)

config ChanA extends TcpIpChCfg_( (* client *)
    variable    = "cA",
    host        = "127.0.0.1", (* ip of the server *)
    port        = "1234"
).

config ChanB extends TcpIpChCfg_( (* server *)
    variable    = "cB",
    host        = "any", (* listens on all hosts *)
    port        = "1234"
).
```

Listing 10: Channel configuration: `hello-world.exdef`

We use the default type ExpiString to model the text message sent over the channel. We define the value it is initialized with by defining a configuration named Msg and setting the key "message" to the desired value, which leads to the final configuration:

```
(* Configuration of the "Hello World" protocol *)

config ChanA extends TcpIpChCfg_( (* client *)
    variable    = "cA",
    host        = "127.0.0.1", (* ip of the server *)
    port        = "1234"
).

config ChanB extends TcpIpChCfg_( (* server *)
    variable    = "cB",
    host        = "any", (* listens on all hosts *)
    port        = "1234"
).

config Msg extends UTF8StringCfg(message = "Hello World!!!").
```

Listing 11: Final configuration: `hello-world.exdef`

Now we can generate a free name c of type Channel⟨ExpiString⟩ in the main process (so that it is visible for both participants) and let it use the configurations we just created:

```
...
process
    new c : Channel@ChanA<ExpiString@Msg>;
    (pB | pA)
```

Listing 12: Creating a communication channel: `hello-world.expi`

This is only the client part of the channel, we need to reconfigure the channel after applying the accept() destructor to model the server side. We do this in the named process pB:

```
let pB =
    let c = accept[->@ChanB](c) in
    0.
```

Listing 13: Channel on the server side: `hello-world.expi`

We proceed with the process pA by generating a new message name m of type ExpiString and configuring it with the configuration Msg. Afterwards, we send the message m on the channel c:

```
let pA =
    new m : ExpiString@Msg;
    out(c, m).
```

Listing 14: Creating and sending the message: `hello-world.expi`

We receive the message in process pB:

```
let pB =
    let c = accept[->@ChanB](c) in
    in(c, m).
```

Listing 15: Receiving the message: `hello-world.expi`

To display the received message, we can use a console channel. We provide a typedef `$STDOUT` that can be used to specify the type of the channel used as standard out. We can display the received message on the console simply by sending it on the console channel:

```
let pB =
    ...
    new cout : $STDOUT;
    out(cout, m).
```

Listing 16: Displaying the message: `hello-world.expi`

The specification of the "Hello World" protocol is complete now:

```
1  (*
2   * A simple "Hello World" protocol
3   *)
4
5  include "../exdef/default.exdef"
6  include "hello-world.exdef"
7
8  let pA =
9      new m : ExpiString@Msg;
10     out(c, m).
11
12 let pB =
13     let c = accept[->@ChanB](c) in
14     in(c, m);
15     new cout : $STDOUT;
16     out(cout, m).
17
18 process
19     new c : Channel@ChanA<ExpiString@Msg>;
20     (pB | pA)
```

Listing 17: Complete specification: `hello-world.expi`

5.2 Code Generation

In order to generate the code, we need to provide several settings to the code generator. We will use the `testCode/expi2java/helloworld` directory as the target directory, and "helloworld" as the package name. The implementation class will be called "HelloWorld". We will use the default code templates in `data/expi2java/javaTemplates`. The output directory must be created first. This results in the following calls:

```
% mkdir testCode/expi2java/helloworld
% ./expi2java -g testData/expi/hello-world.expi \
              -p helloworld \
              -m data/expi2java/javaTemplates \
              -o testCode/expi2java/helloworld \
              -n "HelloWorld"
parsing...                OK
type checking...          OK
inferring types...        OK
```

```
checking configuration... OK
generating code...       OK
```

We set the class path to contain the library and the Bouncy Castle Provider Jar file and compile the generated code as follows:

```
% cd testCode/expi2java
% CLASSPATH=".:../../bin/eXpiLibrary.jar:../../lib/bcprov-jdk16-141.jar"
% javac helloworld/*.java
```

Now we can run the protocol by starting the test application:

```
% java helloworld.Application
Hello World!!!
```

By default, the code of both participants is run in parallel using threads. To see what happens "under the hood" turn on the built-in debug output using the flag `-d`:

```
% java helloworld.Application -d
APP: Starting protocol...
LOG Constructing channel: cA(client, host="127.0.0.1", port=1234,
    timeout=10000, connected=false)
LOG ALIAS cA++ = 1
LOG RUN thread pB
LOG CALL pB: proc ID=3, runonly=0
LOG Constructing channel: TcpIpChannel_24212202(client, host="any", port=1234,
    timeout=10000, connected=false)
LOG Constructing channel: cB(client, host="any", port=1234, timeout=10000,
    connected=false)
LOG ALIAS cB++ = 1
LOG Accepting on: cB(client, host="127.0.0.1", port=1234, timeout=10000,
    connected=false)
LOG RUN thread pA
LOG CALL pA: proc ID=1, runonly=0
LOG called send: cA(client, host="127.0.0.1", port=1234, timeout=10000,
    connected=false)
LOG assigning new socket Socket[addr=/127.0.0.1,port=40393,localport=1234] to
    cB(server, host="any", port=1234, timeout=10000, connected=false)
LOG Accepted: cB(server, host="localhost", port=40393, timeout=10000,
    connected=true)
LOG called receive: cB(server, host="localhost", port=40393, timeout=10000,
    connected=true)
LOG assigning new socket Socket[addr=/127.0.0.1,port=1234,localport=40393] to
    cA(client, host="127.0.0.1", port=1234, timeout=10000, connected=false)
LOG sent: Hello World!!!
LOG END thread pA
LOG received: Hello World!!!
Hello World!!!
LOG ALIAS cB-- = 0
LOG finalizing: cB(server, host="localhost", port=40393, timeout=10000,
    connected=true)
LOG END thread pB
LOG ALIAS cA-- = 0
LOG finalizing: cA(client, host="localhost", port=1234, timeout=10000,
    connected=true)
```

```
38 APP: Protocol completed successfully
39 APP: Bye
```

Listing 18: Debug output of the "Hello World" protocol

As you can see in the log, the protocol starts thread `pB` (line 6), which creates a TCP/IP channel `cB` (line 10) and calls `accept()` on it (line 13). At the same time, thread `pA` starts and starts sending the message (line 17) on the previously created channel `cA` (line 3). Thread `pB` accepts the connection, reconfigures `cB` into a server channel (line 21) and starts receiving the message (line 23). Thread `pA` sends the message and finishes (lines 27-28). Thread `pB` receives it (line 29), prints the message content on the console (line 30) and terminates successfully (line 34).

6 Customizing and Extending Expi2java

6.1 Overview

Expi2java was designed to be easily customizable and extensible. Everything that is involved in the modelling of protocols, code generation and execution of generated protocols can be changed or replaced if needed. The calculus can be extended by adding new constructors and destructors. New types can be easily defined, the behavior of existing types can be adapted using the configurations. The classes implementing every type can also be changed or replaced with customized versions if needed. Even the code generation process can be tweaked by changing the code templates.

However, in most cases the provided set of default types, configurations, constructors, destructors and their implementations can be used without major modifications.

6.2 Adding and Changing Configurations

The configurations provide a set of key-value pairs passed to the implementation classes (see Section 3.1.4). The values are used to set up protocol-specific implementation details, such as IP address and port number of the communication channel, initial value of an integer constant etc. and therefore must be adapted in nearly every protocol. A list of the provided default configurations can be found in Section 3.1.4, page 13.

Changing the values in an existing configuration is easy. It can be done by extending an existing configuration and setting the needed keys. For example, this is the default configuration IntCfg for the type Int:

```
(* signed integer *)
config IntCfg(
    hex       = "yes", (* hexadecimal output in toString() *)
    size      = "4",   (* size in byte, can be in range [1..8] *)
    value     = "0",   (* int is generative, this value is used on generation *)
    type Int(          (* can be used with type Int *)
        class = "Int"  (* implementation class *)
    )
).
```

Listing 19: IntCfg configuration

By default, integers are 32 bit long and are initialized to 0. We can easily create a configuration to generate e.g., an 56 bit integers initialized to 42 as follows:

```
config CustomInt extends IntCfg(
    size    = "7",
    value   = "42"
).
```

Listing 20: Customizing IntCfg

The set of supported keys and values is implementation-specific. If you want to add another key, you will need to extend the corresponding implementation class that can be found in `src/expi2java/src/library/data/*.java`. You can also use another implementation class by adjusting the key "class". See Section 6.5 for more details on extending implementation classes.

6.3 Adding New Types

New types are often needed to describe new data types or express the behavior of new constructors or destructors. Adding new types to the type system is easy, they can be declared using the syntax defined in Table 2 in any included Exdef file.

The user can selectively restrict the generation of fresh names only to the types where it actually makes sense (i.e., where the corresponding implementation is available) in a clean way, using the **generative** keyword. Additionally, when adding a new parametrized type, one should also keep in mind the *subtyping* feature of our type system.

Subtyping Our type system supports *subtyping* with a top type Top. This is similar to the subclassing in object-oriented languages with a single base type (such as Object in Java). By subtyping, any term having type T can also be seen as having any supertype of T. This allows us to omit the information about the nested types where it does not matter and express the types of keys that can encrypt messages of any type, channels that can handle messages of any type etc.

Unlike in Java and other object-oriented languages, in the Extensible Spi Calculus the user can not directly define that a type a subtype of another type. Every type is implicitly a subtype of the top type Top. Additionally, when subtyping instances of the same parametrized type, the parameter types are also taken into account. A parametrized type $T\langle U_1, \ldots, U_n\rangle$ is a subtype of $T\langle U'_1, \ldots, U'_n\rangle$ if every parameter type U_i is related to the corresponding parameter U'_i with respect to its *variance*.

The sense of the subtyping relation is controlled by the variance (see also Pierce, TAPL [Pie02], Chapter 15.2). The relation is reversed for *contravariant* parameters, runs in the same direction for *covariant* parameters and requires the *invariant* parameters to be the same. For example, the type Int (see the list of default types below) is a subtype of Top. The type $\mathsf{SymEnc}\langle T^+\rangle$ is covariant, therefore $\mathsf{SymEnc}\langle\mathsf{Int}\rangle$ is a subtype of $\mathsf{SymEnc}\langle\mathsf{Top}\rangle$. The type $\mathsf{SymKey}\langle T^-\rangle$ is contravariant, therefore $\mathsf{SymKey}\langle\mathsf{Top}\rangle$ is a subtype of $\mathsf{SymKey}\langle\mathsf{Int}\rangle$.

We use `+`, `-` and `0` to denote covariant, contravariant and invariant subtyping in type declarations. The type parameters are invariant by default. The parameters of all user-defined

types can have arbitrary variance, however, the variance needs to be chosen carefully, since the subtyping rules can be used to circumvent the type system on constructors and destructors applications in some cases. We have chosen the variances of the default types (see Section 3.2, page 17) in the way that does not allow such misuses.

Implementation Classes In order to use the new type in the generated Java code, every new type must also be implemented by a new implementation class in the runtime library. The implementation classes can be found in `src/expi2java/src/library/data/` directory. See also Section 6.5 for general information about extending the runtime library.

In order to work with the current code generation process, the implementation classes must obey several rules. Parametric types must be implemented using generic Java classes with the same number of parameters. The implementation classes of non-generative types, such as Bool in Figure 4, should subclass AbstractBase. The implementation classes of generative types, such as Int and KeyPair, should subclass AbstractGenerativeBase. All channels, such as TcpIpChannel, should subclass AbstractChannel and implement methods send() and receive() that are called when some message is sent or received on this channel in the **in** or **out** process.

Every class implementing a generative type should implement a special (Java) constructor that is used to create and initialize an instance of the implementation class by the code generator. This constructor should have the following signature:

```
public ClassName(Map<ExpiString, ExpiString> parameters, ExpiString var) throws EGenerative
```

Listing 21: Special constructor for generative types

Where "ClassName" is the name of the implementation class, "parameters" contain the key-value pairs from the configuration and "var" contains the name of the corresponding Expi variable. Additionally, every implementation class should have a default constructor.

The implementation must reimplement the abstract methods defined in the corresponding abstract superclass and implemented interfaces. The overall class structure of the runtime library can be found in Section 6.5.

6.4 Adding New Constructors and Destructors

Constructors and destructors are defined in Exdef files using the syntax given in Table 2. They are used to perform various operations on data, such as concatenation, encryption or decryption. Every constructor and destructor declaration must be annotated with the type of all arguments and the return type. In case when parametric types are used, a list of all parameters must be given. In order to be used in the generated code, every constructor and destructor must be implemented in the runtime library. The implementation classes can be found in `src/expi2java/src/library/data/*.java`.

We model constructors as static methods in the implementation class of their return type and destructors as (non-static) methods in the implementation class of their first parameter. The implementation functions must have the following signature:

```
// constructor implementation signature
public static Type
nameCtor(Map<ExpiString, ExpiString> params, parameter list) throws EConstructor { ... }

// destructor implementation signature
public Type
nameDtor(Type returnObj, parameter list) throws EDestructor { ... }
```

Listing 22: Signatures of constructor and destructor implementations

Where "name" should be the corresponding constructor or destructor name, "Type" is the return type and "parameter list" is the list of all constructor or destructor arguments. The map "params" will pass the key-value mapping from the configuration of the return type. The argument "returnObj" will pass an instance of the returning object to the destructor, which may be needed in some cases. The specified exception should be thrown on errors, e.g., when the destructor fails.

For example, the implementation of the constructor **fun** succ : (Int) -> Int. from Listing 7 has the following signature:

```
/**
 * Return an integer with the value increased by one
 */
public static Int succCtor(Map<ExpiString, ExpiString> params, Int value)
                                                    throws EConstructor {
    try {
        return new Int(params, value.data+1);
    } catch (EGenerative e) {
        throw new EConstructor("", e);
    }
}
```

Listing 23: Implementation of the succ constructor from `library/data/Int.java`

Universal types of constructors and destructors correspond to generic function arguments. Due to the limitation of the Java type system, all generic parameters must be different even if the corresponding constructor or destructor declaration in the Extensible Spi Calculus has some common types.

The subtyping relation between different parameters must be specified explicitly. For example, the destructor dec from Listing 7 has the type (SymEnc<T>, SymKey<T>) -> T. By subtyping, we can use any subtype of SymEnc<T> as the first and SymKey<T> as the second parameter. Since SymEnc<T> is covariant and SymKey<T> is contravariant, we can decrypt e.g., SymEnc<Hash<Int>> with a key of type SymKey<Top> and obtain a result of type Hash<Top>. In Java, we need to define that the result can be a supertype of the encryption parameter type and a subtype of the key parameter type (IExpi corresponds to the Top type):

```
/**
 * Decrypt an encrypted message
 */
public <U extends IExpi, V extends U, T extends V>
V decDtor(V retObj, SymEnc<T> enc, SymKey<U> key) throws EDestructor {
    ...
}
```

Listing 24: Implementation of the dec destructor

6.5 Extending the Runtime Library

The runtime library is a set of classes implementing all types, constructors and destructors. They can be found in `src/expi2java/src/library/data/*.java`. The name of the implementation class of every term is set using the "class" key in the configuration. Most implementation classes can be used with different types and are controlled by implementation-specific configuration keys. For example, the class SymEnc is an interface to the Java cryptographic providers. It can handle any type of symmetric encryptions supported by the used provider.

We provide a fairly large number of flexible implementation classes for the common cryptographic primitives and data structures to avoid the need of creating new implementation classes. A new class should be needed only in case a completely new type is needed to represent a data type, a cryptographic primitive or a highly customized implementation of an existing type is desirable.

The overall design of the runtime library is shown in Figure 4 (for simplicity, most implementation classes are omitted). The library is composed of several interfaces and abstract base classes, building the core, and implementation classes for the types. The IExpi interface represents the Top type. The interfaces IGenerative, IChannel, IAlias and IKey define a common interface for all generative types, channels, aliases and keys respectively. Abstract base classes AbstractBase, AbstractGenerativeBase, AbstractChannel and AbstractAlias exist for convenience, they contain default implementations of methods used in all subclasses.

The implementation classes, such as Bool, Int and PrivKey<T>, should obey several rules. Implementations of parametric types should be generic with the same order of parameters. Every non-generic implementation class should have a public constructor taking the parameters map:

```
public ClassName(Map<ExpiString, ExpiString> parameters) throws EGenerative
```

Listing 25: Constructor of a non-generic implementation class

Where "ClassName" is the name of the implementation class and "parameters" contain the key-value pairs from the configuration. Every generic implementation class should have a constructor as described above, but with an additional argument for every generic parameter in the same order, at the beginning of the argument list:

```
public ClassName(T type1, ..., Map<ExpiString, ExpiString> parameters) throws EGenerative
```

Listing 26: Constructor of a generic implementation class

Since the default Java serialization (i.e., Serializable and Externalizable interfaces) internally use ObjectInput and ObjectOutput streams that add additional headers to the data, we need to use non-standard serialization interface to achieve the full flexibility. Every implementation class must reimplement methods readFrom() and writeTo() that may encode the data in arbitrary format. Of course, subclasses are free to use default Java serialization internally.

Aliases must be used in cases where several variables in the Extensible Spi Calculus must share the same state in the generated code. For example, we need to use a different channel for

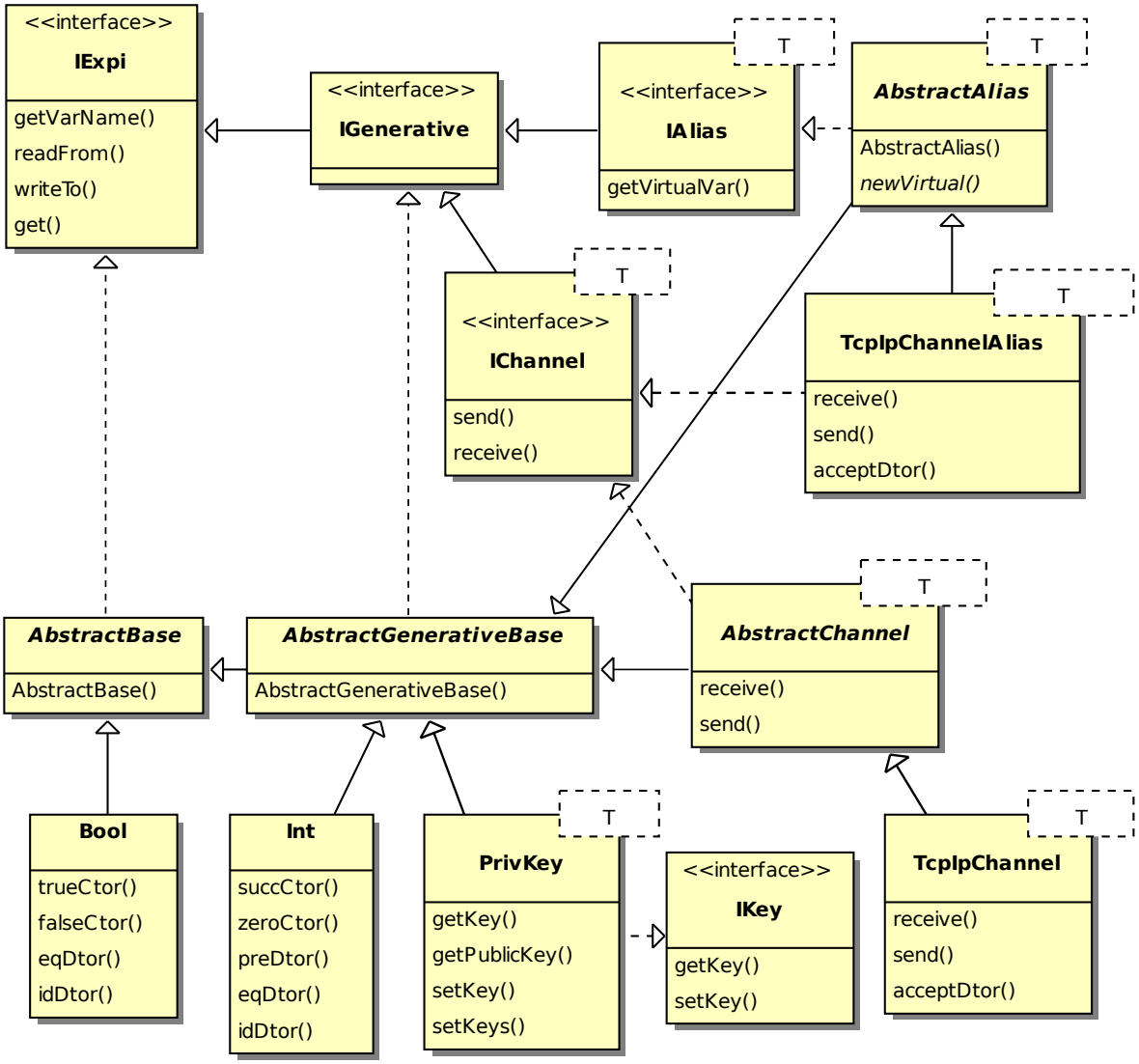

Figure 4: Runtime library

every message of different type in the Extensible Spi Calculus, but if we want to use a TCP/IP channel in the generated code, all these channels must share the same open socket (see also the discussion about channel splitting in Section 3.1.4, page 15). Alias classes can "wrap" other implementation classes and internally link together all variables that have the same value of the "variable" key in the configuration. The configuration should also contain all keys used in the wrapped class.

Internally, aliases share one instance of the wrapped implementation class and forward all method calls to it. Reference counting mechanism is used to forward deinitialize the wrapped object correctly. A restriction of the alias implementation is that they can only wrap generative types and can not be serialized even if the wrapped type can, since they do not have the corresponding constructor (and an uninitialized alias is absolutely useless).

6.6 Adapting Code Templates

Expi2java generates Java code using code templates from `data/expi2java/javaTemplates/` directory. There are several larger templates used to generate files:

`Protocol_%PROTOCOL_NAME%.java`
: Is used to create the classes implementing the protocol

`Thread_%PROC_NAME%_%PROC_ID%.java`
: Is used to create threads implementing parallel composition

`Proc_%PROC_NAME%_%PROC_ID%.java`
: Is used to create classes implementing inlined named processes

`Configurations.java`
: Stores the configuration values used in the protocol

`%APPLICATION%.java`
: Is a simple console application that can run the generated protocol

A list of smaller process code templates from `data/expi2java/javaTemplates/procs/` directory is used to generate every process and term (see Section 3.1.1).

For example, the **let** process is generated using the following template:

```
// %COMMENT%
// %VAR_COMMENT%
%DECLARATIONS%
try {
    %VAR_TYPE% tmp_%PROC_ID% = %DTOR_CODE%;
    %ASSIGNMENTS%

    %PROCESS%
} catch (EDestructor e_%PROC_ID%) {
    Lib.log("destructor does not apply: " + e_%PROC_ID%.getMessage());
    %ELSE_PROCESS%
} finally {
    // cleanup
    %CLEANUP%
}
```

Listing 27: Template `procs/let`

The code is generated by replacing the placeholders (enclosed in % signs) by the corresponding data. The meaning of every placeholder is defined in the class CodeBuilder that can be found in `src/expi2java/src/codegen/`.

The templates can be adapted to some extent without the need to change the code generator. For example, a custom logging can be added into the generated code by adding a few lines to the code templates.

References

[AB05] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005. 1, 3, 6

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998. 3

[BHM08] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Type-checking zero-knowledge. To appear in 15th ACM Conference on Computer and Communications Security (CCS), October 2008. Full version available at `http://www.infsec.cs.uni-sb.de/~hritcu/publications/zk-types-full.pdf`. 8

[Bla08] Bruno Blanchet. *ProVerif v1.14pl4 (Automatic Cryptographic Protocol Verifier) User Manual*. CNRS, Département d'Informatique, École Normale Supérieure, Paris, February 2008. `http://www.proverif.ens.fr/`. 1, 3

[FGM07] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*, pages 31–45. IEEE Computer Society Press, 2007. 8

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002. 28

List of Figures

1 Overall design of Expi2java 2
2 Channel splitting in Perrig-Song protocol 16
3 "Hello World" protocol 22
4 Runtime library 32

List of Tables

1 Grammar of terms and processes 5
2 Grammar of the Exdef declarations 9

Listings

1 Enabling syntax highlighting for Expi files in Vim 4
2 Default constructors and destructors 10
3 AES bundle 12
4 RSA bundle 12
5 Client-server channels setup 16
6 Built-in types 17
7 Default constructors and destructors with types 19
8 Default typedefs 19

9 Simple setup: `hello-world.expi` . . . 23
10 Channel configuration: `hello-world.exdef` . . . 23
11 Final configuration: `hello-world.exdef` . . . 24
12 Creating a communication channel: `hello-world.expi` . . . 24
13 Channel on the server side: `hello-world.expi` . . . 24
14 Creating and sending the message: `hello-world.expi` . . . 24
15 Receiving the message: `hello-world.expi` . . . 24
16 Displaying the message: `hello-world.expi` . . . 25
17 Complete specification: `hello-world.expi` . . . 25
18 Debug output of the "Hello World" protocol . . . 26
19 IntCfg configuration . . . 27
20 Customizing IntCfg . . . 28
21 Special constructor for generative types . . . 29
22 Signatures of constructor and destructor implementations . . . 30
23 Implementation of the succ constructor from `library/data/Int.java` . . . 30
24 Implementation of the dec destructor . . . 30
25 Constructor of a non-generic implementation class . . . 31
26 Constructor of a generic implementation class . . . 31
27 Template `procs/let` . . . 33